8: Failure Tolerant and Reliable Systems

From a previous chapter: *Distributed System:*

- several computational entities ("*nodes*") with their own local memory,
- these entities *communicate* with each other (e.g., by message passing, or by using some joint memory),
- and the system presents itself as a *homogeneous system* to the user (or attempts to do so).

- *A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.*

  – – Leslie Lamport
- *A reliable computer system must be able to cope with the failure of one or more of its components.*

  – – Lamport, Shostak, Pease (1982)

Consider the Cockpit of a Plane

Convention:

- At most n nodes $K_1, \ldots, K_n$, where
- n is known to all,
- and each knows its unique index i.

Core questions:

- How can a Distributed System (DS) maintain reliable operation if some nodes or communication lines fail?
- How can many peer nodes, without a centralized administration server, jointly organize and administrate a DS?

We will consider two types of failures:

Fail-Stop: The failed node just stops working.

Byzantine: The failed node "sends conflicting information to different parts of the system" (Lamport et al.).

Redundancy – indispensable for failure tolerance

- information redundancy (e.g., error-correcting codes)
- backward recovery (go back to a state before the failure; repeat the actions where something failed)
- software redundancy (different and, hopefully, independent implementations of the same functionality)
- physical redundancy (additional hardware components)
  consider a system with n components and required to tolerate up to k failed components:
  - fail-stop: $n \geq k + 1$ suffices
  - byzantine:
    - $n \geq 2k + 1$ with an “infallible” majority decider
    - more if there is no central instance C, of if even C may fail (see below)

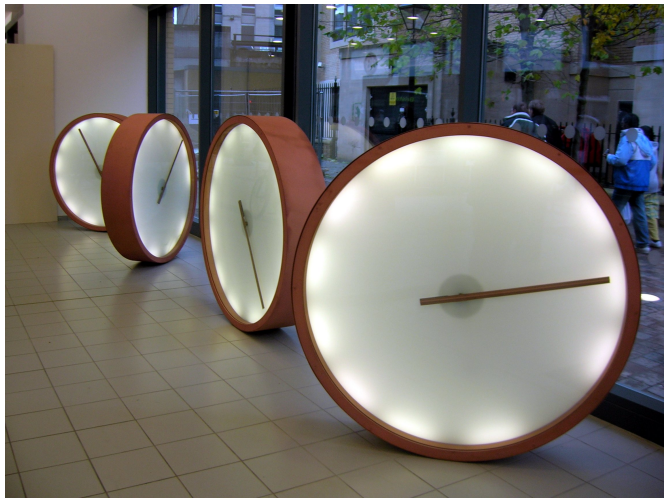
Different modes of communication

- either **synchronous** (upper bounds on the execution time of processes and the transmission time of messages are known, clocks are synchronized, . . . , clocks are synchronized); this is
  - difficult and expensive,
  - but necessary for “hard real-time applications”,
- or **asynchronous** (you never know whether a node has failed or is just a bit slow),
- or **asynchronous with time-out**:
  - no synchronized clocks,
  - but an upper bound on the response time for a node is given.

8.1: Logical Time in Distributed System

Physical clocks:

- Each node has its own clock.
- These clocks are running at slightly different rates.
- Synchronize nodes according to the “physical time”:
  - known (maximal) bias,
  - repeat synchronization frequently.

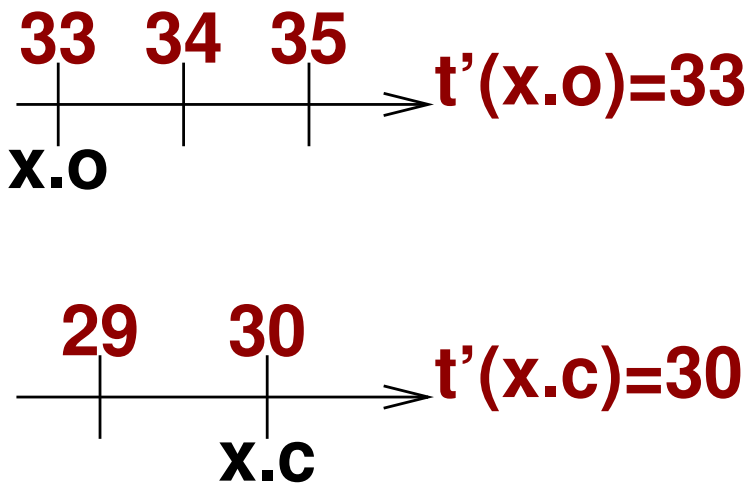


The alternative: “logical clocks”.

Example: A distributed "make"

- specify dependencies between files
- example: **x.o** depends on **x.c**; if **x.o** is either *older* than **x.c** or doesn't exist, **x.c** is recompiled
- the *older* depends on timestamps for **x.o** and **x.c**

- time t_0: **x.o** is generated on node C; timestamp: $t'(x.o)$.
- time $t_1 > t_0$: **x.c** is changed on node E timestamp: $t'(x.c)$.
- If the clocks on C and E are not synchronized, $t'(x.o) > t'(x.c)$ is possible . . . in spite of $t_1 > t_0$!



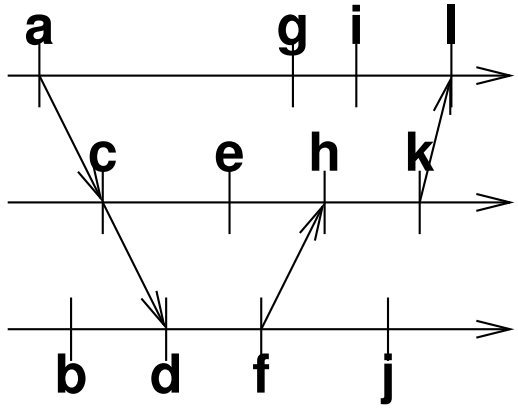
The causal relationship: “happened before”

For events a and b we write $a \rightarrow b$, if *“a happened before b”*.

More formally, for two events a and b, the relationship $a \rightarrow b$ holds if and only if either of the following is true:

1. a and b are on the same node and the physical time of a is before the physical time of b, or
2. a is the event of sending a message, and b is the event of receiving that message at another node, or
3. an event z exists with $a \rightarrow z$ and $z \rightarrow b$ (“transitivity”).

If neither $a \rightarrow b$ nor $b \rightarrow a$ holds, we call a and b “concurrent”; written as “$a||b$”, disregarding the physical time of their occurrence.



Lamport's "logical clocks"

Lamport, 1978

- node K_i: counter C_i $^{(*)}$
- for each local event a_i, we increment the counter $C_i := C_i + 1$ and use the new counter value as the timestamp for a_i
- for the event b_i of receiving a message with timestamp $L(b_i)$, we compute $C_i := \max\{C_i, L(b_i)\} + 1$ and use the new C_i as the timestamp for b_i.

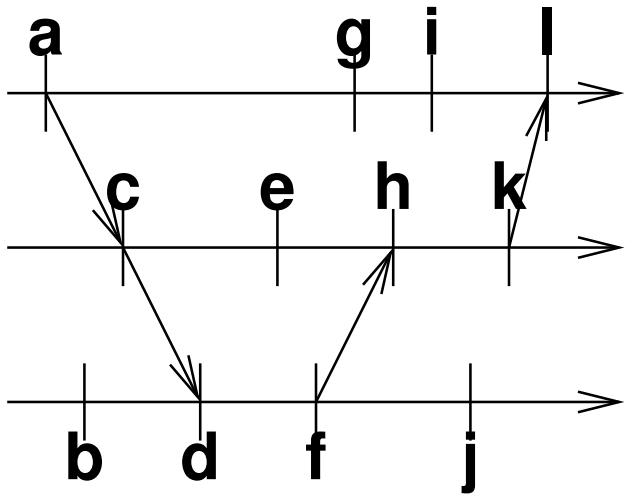
We write $L(x)$ for the Lamport-timestamp of an event x.

Theorem 1

$$(a \to b) \;\Rightarrow\; (L(a) < L(b)).$$

$^{(*)}$ Unfortunately, this assumes that during the lifetime of the system no overflow occurs ... a very dangerous assumption, which frequently causes errors in practice!

Example revisited



- Three nodes, each with current counter 0.
- How do the counters change?

Physical time, Lamport time and causality

1. Unfortunately, the following is wrong:

$$(L(a) < L(b)) \Rightarrow (a \rightarrow b)$$

   Can you find a counter example?
   Why is this a big deal? (Remember the distributed make!)
2. Assume b happened physically *before* a.
   - **2.1** Is $a \rightarrow b$ possible?
   - **2.2** Is $L(a) < L(b)$ possible?
3. Note that concurrency is associative, i.e., $a||b \Rightarrow b||a$.
4. Is concurrency also transitive, i.e., holds

$$(a||b \wedge b||c) \Rightarrow a||c?$$

Totally ordered Lamport time

- L allows the existence of events $a \neq b$ with identical timestamps $L(a) = L(b)$.
- This is only possible if a and b occurred on different nodes K_i and K_j, respectively.
- Sometimes, we need a totally ordered lamport time L':

$$L'(a) < L'(b) \Leftrightarrow \begin{cases} L(a) < L(b) \\ (L(a) = L(b)) \quad \wedge \quad (i < j) \end{cases}$$

Stronger “logical clocks”

What, if we need a logical time T which (in contrast to L and L') actually represents the causal “happened before” relationship?

For all events x we require not only

$$(a \rightarrow b) \;\Rightarrow\; (T(a) < T(b))$$

but also

$$(T(a) < T(b)) \;\Rightarrow\; (a \rightarrow b).$$

- Recall that we have (at most) n nodes. For each nod K_q, the clock is a vector of counters $c^q = (c^q_1, \ldots, c^q_n)$. Intuition:
  - c^q_r is the best lower bound for the "local time" of K_r (to the best of K_q's knowledge)
  - c^q_q is K_q's own "local time".
- For each local event at K_q: $c^q_q := c^q_q + 1$.
- K_q receives a message with timestamp $(T_1, \ldots, T_n)$:
  - For $1 \leq r \leq n$ do $c^q_r := \max\{c^q_r, T_r\} + 1$;
  - $c^q_q := c^q_q + 1$
  - use $c^q = (c^q_1, \ldots, c^q_n)$ as the timestamp for the event of receiving the message

8.2: Distributed Global State

The “state” of a DS is distributed among its nodes and communication channels$^{(*)}$. Our goal:

- Compute a **snapshot** of the state of the DS.
- The state must be **consistent**.

Informally, a **consistent state** is a state in which the DS *could have been* without violating the “happened before” relationship. It need not actually have been in that state. Such snapshots are useful

- for garbage collections in a DS,
- to reclaim unused resources in a DS,
- to detect a deadlock in a DS,
- ...

$^{(*)}$ K_i has sent a message to K_j, but K_j has not yet received it.

The “Cut”

- At any point of time, the *local state* of any node K_i is defined by a finite series of events $E_k^i = (e_1^i, e_2^i, \ldots e_k^i)$. $^{(*)}$
- A *Cut* describes the *global state* as the union of the n local states

$$E_{k_1}^1 \cup E_{k_2}^2 \cup \cdots \cup E_{k_n}^n.$$

- The union of the most recent states is the *frontier* of the cut:

$$\left\{ e_{k_1}^1, e_{k_2}^2, \ldots, e_{k_n}^n \right\}.$$

$^{(*)}$ … and the initial state of K_i, which we assume to be known.

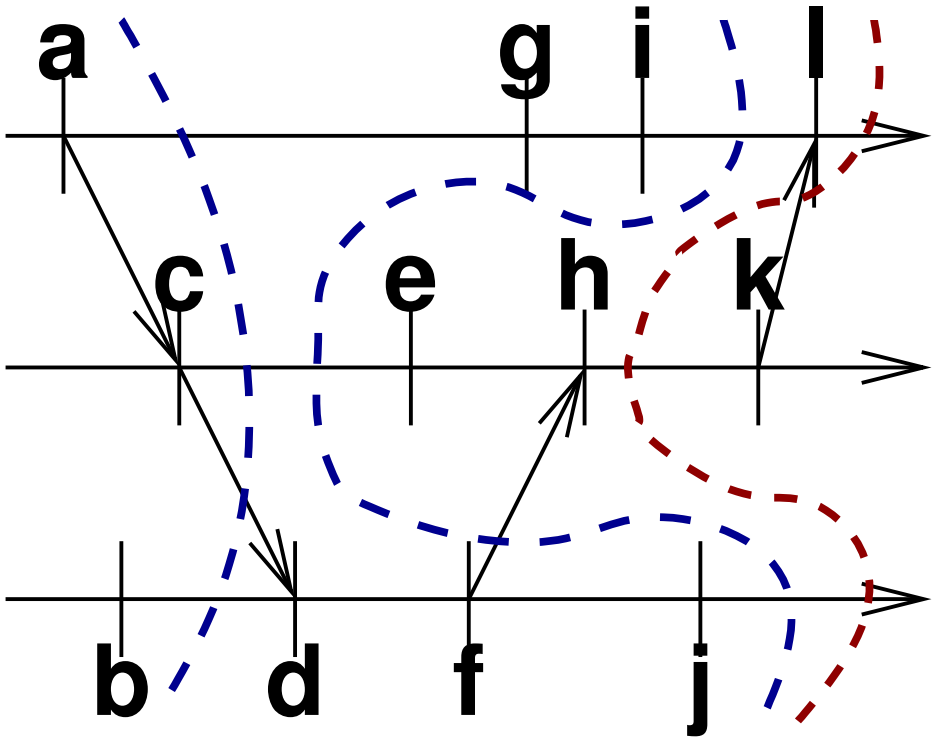
Consistency of cut and state

- A cut C is *consistent*, if and only if

$$\forall b \in C : (a \rightarrow b) \Rightarrow (a \in C).$$

- A cut is *strongly consistent*, if for all events $a \in C$ which represent sending a message to another node the event b of receiving that message is also in C (i.e., the message is no longer in transition).
- A global state is *(strongly) consistent*, if and only if it is defined by a *(strongly) consistent* cut.

Examples



- an *inconsistent cut* with the frontier (**l**, **h**, **j**) and
- two *consistent cuts* with frontiers (**a**, **c**, **b**) and (**i**, **c**, **j**).
- Neither (**a**, **c**, **b**) nor (**i**, **c**, **j**) is strongly consistent.
- Cuts like (**a**, **c**, **d**) and (**a**, **d**, **e**) would be *strongly consistent*.

The Chandy-Lamport Algorithm

Strongly consistent snapshots: Lamport, Chandy, 1985

Requirements

- Reliable nodes (during the time of computing the cut)
- Connections between nodes:
  - a complete directed graph,
  - FIFO in each direction, and
  - reliable (each message sent is received exactly once).
- An arbitrary node K_x (the “observer”) initializes the snapshot.
- While the snapshot is taken, the DS can continue to work normally.

The idea for the algorithm

- The observer K_x sends "snapshot" to all other nodes.
- When K_i receives it first "snapshot" (from K_x, or maybe from another node), it sends out "snapshot" to each K_j, $j \neq x$.
- Thus, there will be *one "snapshot" message* passing through each communication channel.
- All messages sent from K_i *before the "snapshot" message* determine the consistent state and thus have to be stored at the receivers' side.
  Messages sent out *after the "snapshot" message* are not part of the consistent state.
- Thus, K_i will store all messages from K_j until it receives the "snapshot" from K_j.
- Ultimately, $K_i \neq K_x$ will send all messages the internal state it had stored, and all stored messages, to K_x, and K_x will output the cut.

The algorithm (start)

- The Observer K_x
  1. stores its own local state into a variable Z_x,
  2. defines an (initially empty) message list $M(k)$ for each $k \in \{1, \ldots, n\}, k \neq x$, and
  3. sends a snapshot message to all K_j, $j \neq x$.
- Each node $K_i \neq K_x$ receiving its first snapshot message from any K_j (either $j = x$ or $j \neq x$):
  1. stores its own local state into a variable Z_i,
  2. sends a snapshot message to all K_j, $j \neq i$, and
  3. defines an (initially empty) message Lister $M(k)$ for each $k \in \{1, \ldots, n\}, k \notin \{i, j\}$.

The algorithm (main part)

Assume K_i has already sent out its snapshot messages.

Now it receives a message M from K_m. We distinguish

Either M is a normal, or it is a snapshot message.
Either the list $M(m)$ exists, or it has been deleted.
} 4 cases

	M normal	M snapshot message
$M(m)$ deleted	**null**;	**raise** Program_Error **with** Debug_Info;
$M(m)$ exists	append M to $M(m)$	1. add the events from $M(m)$ to Z_i; 2. delete $M(m)$; 3. **if** no lists are left **then** send Z_i to the observer K_x (or mark Z_i as done if $i = x$) **end if**;

The algorithm (finish)

Each $K_i \neq K_x$ is done with the snapshot, when it has sent its Z_i to the observer K_x.

K_x waits for n lists Z_i (including its own).

The union of these Z_i is a strongly consistent cut.

And why is this cut strongly consistent?

8.3: More Distributed Algorithms

The easy approach: A central coordinator C:

Three types of messages:

- **request** X to C – – I want to use X
- **grant** X from C – – you can exclusively use X
- **release** X to C – – I don't need X any more

C maintains a queue for each shared resource X.

Making the system reliable, even though C may fail:

- send **ping** to C, when waiting too long
- in the case of a **time-out**: the peers need to elect a new coordinator

Ricart-Agrawala without token

No central coordinator!

Requirements:

- complete directed graph (each node K_i can directly communicate with each other node K_j)
- total logical clocks / timestamps

The algorithm

- 3 states: released$_i(X)$, requested$_i(X)$, and held$_i(X)$
- initially: released$_i(X)$ for all K_i
- invariant for each X: at most one i is held$_i(X)$
- when a node K_i wants to reserve X:
  - state becomes requested$_i(X)$;
  - send **request** X to all nodes K_j, $j \neq i$;
  - if a **request** X from K_ℓ arrives with a smaller timestamp: send **grant** X to K_ℓ
  - when *all* other nodes did **grant** X to K_i: state becomes held$_i(X)$;
  - when X is no longer used: turn state into released$_i(X)$;
  - if there are one or more **request**s pending: **grant** X to K_j, where K_j is the node whose **request** has the smallest timestamp
- requires each node to maintain a queue for each X

Is Ricart-Agrawala more reliable / better than using a central coordinator?

+ When the coordinator fails, access to X is blocked until a new coordinator has been elected.
– For R-A: access to X is blocked when *any* node fails (until all nodes still alive agree on the list of life nodes).
– R-A needs $2 * (n - 1)$ messages, the central coordinator approach only 3.

$\Rightarrow$ The central coordinator approach seems to outperform the "real" distributed algorithm (. . . but we still have to discuss how to elect a new coordinator).

Electing a Coordinator

- When a coordinator C fails, any node K_i can take over.
- It doesn't matter which K_i takes over – but all nodes still alive must agree on the same K_i.
- If the same K_i is chosen for too many different coordination tasks, it is allowed to delegate tasks to other nodes.

The setting:

- Each node K_i has a unique index $i \in \{1, \ldots, n\}$ (as above).
- Nobody knows, which $K_1, \ldots, K_n$ are actually running.
- The "Bully"-algorithm will select the node with the largest index $i \in \{1, \ldots, n\}$.

The “Bully” Algorithm

1. When K_i comes to the conclusion that C has failed,
   - it sends an **election** message to all nodes $K_{i+1}, \ldots, K_n$,
   - waits for answers, and
   - if there is no answer (time-out), K_i becomes the new coordinator (and tells this all the other nodes),
   - else it waits until a node K_j with $j > i$ claims to be the new coordinator.
2. When K_j receives an **election** message from K_i, $i < j$
   - it sends an answer to K_i
   - and continues as in **1.**

Token-based Algorithms

- Exactly one node owns the “token” and is allowed to use X.
- If one wants to use X, one has to wait for the token.
- Using X is only allowed after having received the “token”.

Token-based Ricart-Agrawala

- totally ordered timestamps (once again)
- an arbitrary node owns the token
- If K_i needs X and doesn't own the "token":
  - it sends a **request** with its current logical clock as timestamp to all nodes K_j, $j \neq i$
  - and waits for the "token".
- All nodes maintain a queue of **request** messages.
- When the "token" owner doesn't need X any more, and the queue isn't empty,
  - it sends the "token" to the node whose **request** has the smallest timestamp.

Token-Ring

- no timestamp
- logical ring (e.g., $K_1 \rightarrow K_2 \rightarrow \cdots \rightarrow K_n \rightarrow K_1$; not necessarily related to the actual network topology);
- when K_i receives the token
  - it either forwards it to the next in the ring, immediately,
  - or it uses X and forwards the token later.

Token-based algorithms: advantages and disadvantages

R.A.-Token:

- \+ failure of a node doesn't hurt, except when this node happens to own the "token"
- – n messages ($n-1$ **request**s and 1 **grant**)
- – each node has to maintain a **request** queue
- – a reliable system must prepare for the loss of a token

Token-Ring:

- – failure of a node results in the "token" being lost
- \+ no messages to be granted access to X (just wait for the token)
- – but many messages are sent around, even at times when nobody needs X
- \+ very simple
- – a reliable system must prepare for the loss of a token

8.4: Byzantine Faults without “Infallible” nodes

Problem 1 – the byzantine generals:

- n divisions of the Byzantine army besiege a hostile castle.
- Each division has its own general; $k < n$ of the generals are traitors. The generals communicate to each other by messengers giving oral messages.
- Decision of the day: *retreat* or *attack*.
- All loyal generals must either jointly *retreat* or jointly *attack*; if some give different commands then others, the enemy wins.

Problem 2 the byzantine general and his lieutenants:

- One commanding general and $n - 1$ lieutenants, each commanding one division of the army.
- $k < n$ of the officers can be traitors.
- The general sends a *retreat* or *attack* to his lieutenants.
- No “infallible” node: even the general can be a traitor, in which case he may send send *retreat* to some of his lieutenants and *attack* to others.
- After some further communication all loyal officers must either jointly *retreat* or jointly *attack*.

Results for k traitors

theory: Lamport, Shostak, Pease (1982),
practical solution: Castro and Liskov, (1999)

- If there is a solution for problem 2, there is also a solution for problem 1, and vice versa ($\rightarrow$ blackboard)
- $k = 1$: three generals are insufficient ($\rightarrow$ blackboard)
- $k = 1$: four generals are sufficient ($\rightarrow$ blackboard)
- any k: $3k + 1$ generals are necessary and sufficient
- any k: when using unforgable signatures rather than oral messages, $k + 2$ generals (two or more loyal ones) suffice.